WO0079386

Publication Title:

METHOD FOR PROVIDING AN OPTIMIZED MEMORY ALLOCATION FOR JAVA FUNCTIONS

Abstract:

Abstract of WO0079386

A method for operating a computer system (100, 200) to provide an optimized memory allocation (102, 121, 122) for Java functions (112) comprises the steps of (i) reading a Java program (240, 250) with classes and interfaces having functions within the classes to establish an interference graph (300) with nodes (301-305) for each interface and inter-node edges (331-335) for the occurrence of first and second interfaces within a single class; and (ii) assigning unique offsets (e.g., <i>Offset</i>=0) to pluralities of interfaces (e.g., 1 and 3) which are not connected by said edges; and (iii) storing corresponding pointers (129, e.g., PFT 1, PFT 3) in the memory (102) in sections (121) for each class wherein each pointer in the plurality is identified by the previously assigned offset. Data supplied from the esp@cenet database - Worldwide

---

(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
28 December 2000 (28.12.2000)

PCT

(10) International Publication Number
WO 00/79386 A1

(51) International Patent Classification[7]: G06F 9/44

(21) International Application Number: PCT/RU99/00210

(22) International Filing Date: 22 June 1999 (22.06.1999)

(25) Filing Language: English

(26) Publication Language: English

(71) Applicant *(for all designated States except US)*: **MOTOROLA, INC.** [US/US]; 1301 E. Algonquin Road, Schaumburg, IL 60196 (US).

(72) Inventors; and

(75) Inventors/Applicants *(for US only)*: **KIRILLIN, Vyacheslav** [RU/RU]; pr. Marshala Kazakova, 10-910, St.Petersburg, 198302 (RU). **PREOBRAZHENSKY, Dmitry** [RU/RU]; ul. Manchesterskaya, 12-56, St.Petersburg, 194156 (RU). **KLEMENTIEV, Oleg** [RU/RU]; pr. Dunaisky, 42/79-1-380, St.Petersburg, 192281 (RU).

(74) Agents: **RYBAKOV, Vladimir, M.** et al.; Agency of Patent Attorneys "Ars-Patent", Shvedsky per., 2-314, P.O.B. 230, St.Petersburg, 191186 (RU).

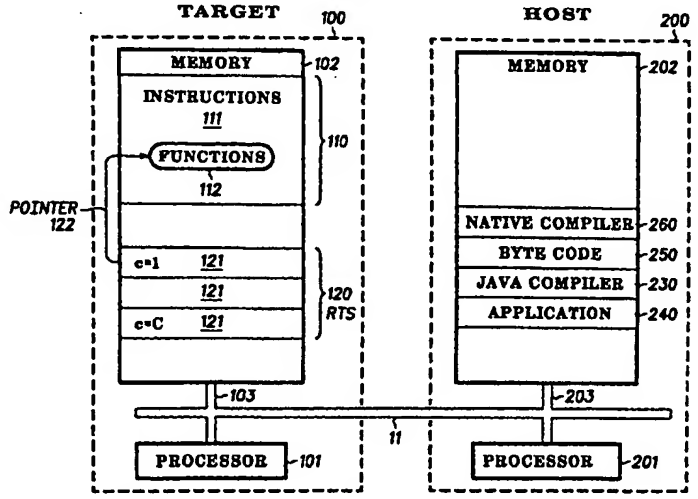
(81) Designated States *(national)*: JP, US.

(84) Designated States *(regional)*: Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE).

Published:
— *With international search report.*

For two-letter codes and other abbreviations, refer to the "Guidance Notes on Codes and Abbreviations" appearing at the beginning of each regular issue of the PCT Gazette.

(54) Title: METHOD FOR PROVIDING AN OPTIMIZED MEMORY ALLOCATION FOR JAVA FUNCTIONS



10

(57) Abstract: A method for operating a computer system (100, 200) to provide an optimized memory allocation (102, 121, 122) for Java functions (112) comprises the steps of (i) reading a Java program (240, 250) with classes and interfaces having functions within the classes to establish an interference graph (300) with nodes (301-305) for each interface and inter-node edges (331-335) for the occurrence of first and second interfaces within a single class; and (ii) assigning unique offsets (e.g., *Offset*=0) to pluralities of interfaces (e.g., 1 and 3) which are not connected by said edges; and (iii) storing corresponding pointers (129, e.g., PFT 1, PFT 3) in the memory (102) in sections (121) for each class wherein each pointer in the plurality is identified by the previously assigned offset.

WO 00/79386 A1

METHOD FOR PROVIDING AN OPTIMIZED MEMORY ALLOCATION FOR JAVA FUNCTIONS

Field of the Invention

The present invention generally relates to computer systems and methods to operate them, and, more particularly, relates to a Java computer system, a storage medium and an apparatus to optimize addresses.

Background of the Invention

The programming language "Java" found widespread use in software applications which control the operation of different hardware (e.g., computer controlled household appliances) without the need to be rewritten. In other words, a single Java application runs on different target computers (e.g., microprocessors of different manufactures).

However, this feature requires control software which is memory consuming and which eventually delays the execution of the application at run-time. It is an object of the present invention to provide an improved method to operate a computer which mitigates the disadvantages of the prior art.

Brief Description of the Drawings

FIG. 1 illustrates a simplified block diagram of computer system in which the present invention can be embodied;

FIG. 2 illustrates a simplified pseudo source code listing for a Java application;

FIG. 3 illustrates a simplified block diagram with a partial view of a memory having runtime structures divided into sections;

FIG. 4 illustrates a simplified block diagram of a plurality of run-time structures for a plurality of classes according to the prior art;

FIG. 5 illustrates a simplified block diagram of a plurality of run-time structures for a plurality of classes according to the present invention;

FIG. 6 illustrates a simplified interference graph according to the present invention by example;

FIG. 7 illustrates a simplified interference graph in an inversion of the graph of FIG. 6; and

FIG. 8 illustrates a simplified flow chart diagram of a method in a preferred embodiment of the present invention.

Detailed Description of a Preferred Embodiment

The present invention is explained in connection with the programming language Java, but this is not limiting. Those of skill in the art are able, based on the description herein, to apply the present invention to other languages without departing from the scope of the present invention. For convenience, a glossary of terms is provided prior to the claims.

FIG. 1 illustrates a simplified block diagram of computer system 10 in which the present invention can be embodied. For convenience, system 10 is illustrated by target computer 100 and host computer 200 (dashed boxes) having different functionality. However, computers 100 and 200 can physically be implemented by the same hardware.

Target computer 100 is a computer on which an application is executed (at application run-time). For example, target computer 100 controls the operation of a household appliance. As every computer, target computer 100 has target processor 101 which is controlled by target instructions ("native instructions" or "machine instructions") stored in a target memory 102 (coupled by bus 103). Since target computer 100 is often a low-price mass product, processor 101 and memory 102 should be as small as possible.

Host computer 200 is a computer needed to provide target computer 100 with target instructions 111 (e.g., in section 110 of memory 102). Similar as above, host computer 200 has host processor 201 coupled to host memory 202 by bus 203.

For convenience of explanation, bus 11 between busses 103 and 203 couples computers 100 and 200. Preferably, this connection is provided only temporarily to transfer data from computer 200 to computer 100 (also vice versa). Persons of skill in the art are able to provide data transfer by other means, such as, for example, via further memory (hard and floppy disks, CD-ROMs, tapes).

Java compiler 230 reads application software 240 in its source code representation (e.g., files with extension ".java"). Java compiler 230 preferably does not consider the type of the target processor. Java compiler 230 usually operates on host computer 200 (e.g., in memory 202), but this is not essential. Java compiler 230 - optionally - provides intermediate representation 250 of application software 240 in a so-called bytecode representation (e.g., files with extension ".class"). Preferably, the bytecode is a sequence of intermediate instructions each having an opcode (one byte, e.g., "add") and having zero or more operands (e.g., variables).

Native compiler 260 reads software in its bytecode representation 250 and provides target instructions 111. Native compiler 260 operates on host computer 200 (as illustrated) or on target computer 100. Java compiler 230 and native compiler 260 can be combined into a single compiler. Native compiler 260 has to be implemented for each available target processor 101.

Native compilers are classified (a) in static compilers or (b) Just-In-Time (JIT) compilers. The present invention will be explained by example for compiler 260 being a static compiler. Persons of skill in the are able, based on the description herein, to apply the present invention also for other types.

JIT compilers can be considered as dynamic compilers because they operate in close cooperation with (c) Virtual Machines (VM). The classification depends on the computer where the native compiler operates (host 200 or target 100), the operation time (before or during application runtime), and on other criteria.

(a) The static compiler (e.g., compiler 260) operates on host computer 200 and provides target instructions 110 which are communicated to target memory 103 by so-called executable files (e.g., through bus 11). Static compiler 260 operates before application run-time.

(b) The JIT compiler operates on target computer 100 at application run-time and translates bytecode instructions (e.g., also stored in target memory) into target

instructions 111. The JIT also provides predefined instructions sequences (functions) to the target memory at application run-time.

(c) The VM operates on target computer 100 like an interpreter and provides target instructions 111 by decoding bytecode instructions into target instructions at application run-time.

Functions 112 are pluralities of predefined instruction sequences (of different lengths) stored in target memory 102. As illustrated in FIG. 1, functions 112 have the form of target instructions 111. It is also possible to define functions in the bytecode instructions (not illustrated).

Section 120 in memory 102 stores so-called run-time structures RTS(c) (reference number 121) for a total number of C classes (having index c). Each RTS(c) has (one or more) pointers 122 to functions 112 in section 110. Preferably, each RTS(c) has only pointers to instructions belonging to a single class.

The present invention provides a method to operate a computer (i.e., host computer 200) to provide RTS(c) in such an optimization that the access to target instructions 111 by target processor 101 is faster and less memory consuming than with prior art structures. For convenience, it can be assumed that a static compiler (e.g., compiler 260 in FIG. 1) receives application 240 (e.g., in source code of FIG. 2) and provides target instructions 111 to memory portion 110 in FIG. 1.

FIG. 2 illustrates a simplified pseudo source code listing for a Java application (e.g., application 240) by way of example FIG. 2 identifies:

- functions by the word "*function*" and indices "i" (interface index) and "f" (function index);
- interfaces by the word "*interface*" and index "i"; and
- classes by the word "*class*" and index "c". The example of FIG. 2 uses a total number of I = 5 interfaces, a total number of C = 6 classes and assumes that each interface implementation only has F = 2 functions (index f = 1, 2). The keyword "*implements*" indicates that a class uses functions of a single interface (e.g., classes 1, 3 and 6) or of multiple interfaces (e.g., classes 2, 3, and 5).

In source code using proper syntax, functions are often represented by identifiers which are independent from a class (see below). For example, the function "X.println( )" sends the value of an object "X" to a printer (e.g. via bus 11). The

corresponding bytecode instruction is "invokeinterface println" with paramenter "X" on an execution stack. In target memory 102, functions are implemented by target instructions 111 forming functions 112 (cf. FIG. 1). The literature relating to Java describes functions by the term "method".

Interfaces are pluralities of functions (e.g., F = 2 functions for each interface). For example, an output interface comprises functions (e.g., print to printer, write to display) which send data from target computer 100 to a peripheral device (e.g., the printer or the display). Interface implementations are pluralities of function implementations.

Classes define properties of data and functions. For example, a first class (e.g., index c = 1) defines integer variables (i.e., 0, 1, 2, 3...) and a second class (e.g., index c = 2) defines characters (i.e., "a ", "b ", "c ",...).

A function can be defined for a single class or for multiple classes. When a function is defined for multiple classes, then target instructions 111 are different for each class. Assume, for example, that the function "X.println( )" is defined both for the class 1 (integer) and for the class 2 (characters). Target instructions 111 for printing integer variables and for printing characters are different (i.e., different pointers 122 from different RTS). In other words, the function has a first implementation (target instructions for the first class) and has a second implementation (target instructions for the second class). Depending to which class an object "X" belongs to, the target processor uses different target instructions (hereinafter "function implementations") for the same function (as mentioned above having class-independent identifier).

For example, the target processor executes the above mentioned function "X.println( )" with first target instructions when variable X stores an object of the integer class or with second, different target instructions when X stores an object of the character class. Java allows to define a variable X as a variable of the integer class or of the character class if both classes support an output interface. An object indicates an actual value stored, for example, in a variable.

It is an advantage that target processor 101 accessing memory structure 120 of the present invention loads suitable target instructions 111 faster than a target processor accessing prior art structures.

The number of interfaces implemented by a given class is referred to as $I_{CLASS}(c)$ with, for example (FIG. 2):

$$I_{CLASS}(1) = I_{CLASS}(3) = I_{CLASS}(6) = 1, \quad (1)$$
$$I_{CLASS}(3) = I_{CLASS}(5) = 2 \text{ and}$$
$$I_{CLASS}(2) = 3.$$

A class using two or more interfaces is referred to by the term "multiple interface class", abbreviated "MIC". The example of FIG. 2 illustrates a total number of J = 3 MIC 2, 3 and 5.

FIG. 3 illustrates a simplified block diagram with a partial view of memory 102 having RTS(c) (a single class c) divided into section 125 and section 126. Section 125 is further divided into S slots 127. Section 126 is further divided into further slots 128 for function tables. Slot 127 comprises pointer 129 to the function tables (hereinafter "pointer to function table - PFT") stored in slots 128. One PFT only points to the function tables for a single interface. In the following, two or more PFTs are distinguished by interface index i.

Slot 127 can be empty, but this should be avoided. The function tables have pointers 122 to target instructions 111 (cf. FIG. 1). Although convenient, splitting the RTS(c) into function tables is not essential for the present invention. Therefore, RTS(c) is further discussed or illustrated only in connection with the PFT in slots 127. Each slot is identified by an offset counter "*Offset*" preferably having values between

$$Offset = 0 \ldots S\text{-}1 \quad (2)$$

In other words, *Offset* is the address by which the PFT is identified. According to the present invention, the number S of slots (e.g., S = 3) is smaller than the total number I of interfaces (e.g., I = 5). This is a major advantage over the prior art. The following FIGS. illustrate how this can be achieved.

FIG. 4 illustrates a simplified block diagram of a plurality of run-time structures (RTS(1) to RTS(6) for classes 1 to 6, "C1" to "C6") according to the prior art. For convenience, RTS(1) to RTS(6) are illustrated next to each other so that slots 127 having equal *Offset* (0...4) are adjacently placed. Each RTS(c) has S = I slots, resulting in a total number of S * C = I * C slots (e.g., 5 * 6 = 30).

The first slot row (*Offset* = 0) stores only PFTs for interface 1 ("PFT 1"), the second slot row (*Offset* = 1) stores only PFTs for interface 2 and so on. Or, looking at

columns, RTS(1) for class 1 only stores the PFT for interface 1 (cf. statement (21) in FIG. 2), RTS(2) for class 2 only stores the PFTs for interfaces 1, 2, and 4 (cf. (22) in FIG. 2) and so on. From the total number of 30 slots, 10 slots are used to store PFTs and 20 slots are empty, resulting in a used-to-total ratio 10 to 30, or 33%.

FIG. 5 illustrates a simplified block diagram of a plurality of run-time structures (RTS(1) to RTS(6) for classes 1 to 6, "C1" to "C6") according to the present invention. Similar as in FIG. 4, RTS(1) to RTS(6) are illustrated next to each other so that slots 127 having equal *Offset* (values 0...2) are adjacently placed. Persons of skill in the art can place RTS(c) otherwise, for example, arbitrarily in a heap. Each RTS(c) has only $S' < I$ slots, resulting in a total number of $S' * C$ slots (e.g., $3 * 6 = 18$). From the total number of 18 slots, 10 slots are used to store PFTs and only 8 slots are empty, resulting in an higher used-to-total ratio 10 to 18, or 55%.

Different to the prior art, the first slot row (*Offset* = 0) stores the PFT for interface 1 and for interface 3, the second slot line (*Offset* = 1) stores the PFTs for interface 2 and 5; and the last row stores the PFTs for interface 4. Looking at the columns, RTS(c) store the tables as above.

It will now be explained how PFT offsets are assigned to the interfaces.

(i) According to a first assignment rule of the present invention, sets of interfaces which do not belong to common classes (i.e. equal index c) have the same offset. For example, a first set comprises interfaces 1 and 3 which belong to different classes (cf. FIG. 2, interface 1 to classes 1, 2 and 5 and interface to classes 3 and 4). The first set has been assigned to *Offset* = 0. A second set comprises interfaces 2 and 5 which also belong to different classes (cf. interface 2 to classes 2 and 3 and interface 5 to class 5). The second set has been assigned to *Offset* = 1. Finally, a third set with the single interface 4 has been assigned to *Offset* = 2. Among other combinations, it is also possible, for example, to assign *Offset* = 1 to the first set (interfaces 1 and 3) and to assign *Offset* = 0 the second set (interface 2 and 5).

(ii) According to a second assignment rule, applied optionally, the smallest offset (e.g., *Offset* = 0) is assigned to that interface set (e.g., interfaces 1 and 3) implemented by most of the classes (e.g., by 5 classes 1-5 out of C = 6).

(iii) According to a third assignment rule, also applied optionally, the smallest offset (e.g., *Offset* = 0) can be assigned to that interface set (e.g., interfaces 1 and 3)

which comprises the highest number of interfaces in a set (in the example, sets with 2 interfaces).

It is convenient for explanation to discuss the offset assignment with the help of a graphical representation.

FIG. 6 illustrates simplified interference graph 300 according to the present invention by example (cf. FIG. 2). Graph 300 has nodes 301 to 305 corresponding to interfaces 1 to 5, respectively, and has edges 331-335 between some pairs of nodes. It is understood that graph 300 is a mere representation for the convenience of explaining an analysis according to the present invention. Persons of skill in the art can perform such an analysis by a computer (e.g., host computer 200) without displaying a graph.

Edge 331 at node pair 301/302 represents the occurrence of interfaces 1 and 2 in statement (22); edge 332 at node pair 302/304 represents interfaces 2 and 4 in the same statement (22); edge 333 at node pair 301/304 represents interfaces 1 and 4 in the same statement (22); edge 334 of node pair 302/303 represents interfaces 2 and 3 in statement (23); and edge 335 of node pair 301/305 represents interfaces 1 and 5 in statement (25).

FIG. 7 illustrates simplified interference graph 300' in an inversion of graph 300. Similar as above, graph 300' has nodes 301' to 305' corresponding to interfaces 1 to 5, respectively. Graph 300' has edges 320, 321, 322 and 323 between pairs of nodes where graph 300 of FIG. 6 does not have edges. Edge 320 (nodes 301', 303') indicates that interfaces 1 and 3 are independent so that they can be assigned a common offset (e.g., *Offset* = 0). Edge 321 (nodes 302', 305') indicates that interfaces 2 and 5 are independent so that they also can be assigned a further common offset (e.g., Offset = 1). Edges 323 and 322 indicate the independence of interface 4 to interfaces 3 and 5.

The problem of assigning offsets can be considered as a graph-coloring problem when different offsets are considered as different colors of the nodes of graph 300. Such problems can be solved by procedures well known in the art. The solution proposed in connection with FIG. 8 is intended to illustrate a preferred embodiment of the present invention. However, persons of skill in the are able to solve the graph-coloring problem by other means.

The present invention is now described as a method for providing a run-time structure (e.g., RTS(c)) for an application (e.g., Java) to a memory (e.g., memory 102 of target processor 100). The steps are (i) reading a code representation (e.g., source code 240, byte code 250) of the application in which interfaces are used by classes (cf. FIG. 2, "*class ... implements interface ...*"); and (ii) writing, separate for each class, PFTs to memory locations (e.g., slots 127) which are identified by interface offsets (e.g., *Offset*) within the classes (e.g., within RTS(c) for class c). The method characterized in that • reading comprises identifying at least a first set of first and second interfaces (e.g., interfaces 1 and 3) such as there is no class which uses together first and second interfaces of the first set (e.g., classes 1 to 6 do not use interfaces 1 and 3 together), and • writing comprises assigning a first common offset (e.g., *Offset* = 0) to corresponding first and second PFTs (e.g., PFT 1, PFT 3 in FIG. 5) which belong to the first and second interfaces, respectively, of the first set.

By repeating reading and writing for a second set of further interfaces (e.g., interfaces 2 and 5) with a second common offset (e.g., *Offset* = 1) the method, preferably, assigns the smallest offset (e.g., first common offset) to that set (set with interfaces 1 and 3) which is used by most of the classes (e.g., by 5 classes 1 to 5).

In other words, the present invention describes a method for operating a computer system (e.g., system 10 with computers 100 and 200) to provide an optimized memory allocation (e.g., target memory 102, run-time structure 121, pointer 122) for Java functions (112) which comprises the steps of (i) reading a Java program (240, 250) with classes and interfaces having functions within the classes (cf. FIG. 2) to establish an interference graph (300) with nodes (301-305) for each interface and inter-node edges (331-335) for the occurrence of first and second interfaces within a single class; and (ii) assigning unique offsets (e.g., *Offset* = 0) to pluralities of interfaces (e.g., 1 and 3) which are not connected by the edges; and (iii) storing corresponding PFTs (e.g., PFT 1, PFT 3) in the memory (e.g., memory 102) in sections (e.g., RTS 121) for each class wherein each PFT (i.e., PFT 1 and PFT 3) in the plurality is identified by the previously assigned offset.

FIG. 8 illustrates a simplified flow chart diagram of method 400 in a preferred embodiment of the present invention. For convenience of explanation, it is assumed that the steps are performed by host computer 200 being controlled by compiler 260.

Persons of skill in the art can, based on the description herein, perform method 400 by other means without departing from the scope of the present invention. According to method 400, computer 200 analyzes the source code for the occurrence of interfaces, identifies interface sets, and assigns offsets to the interface sets. Method 400 comprises the following steps: start 401, numbering classes 410, bit vector defining step 420, 425, interference checking step 430 (query with YES 431, NO 432), offset assigning step 440, merging step 450, 460, and end 499. Steps 425 and 430 in combination form identifying interface sets step 435 (dashed frame).

Preferably, computer 200 executes method 400 in the following order: start 401, numbering classes 410; bit vector defining 420 and 425; interference checking 430, NO 432, offset assigning 440, merging 450 (or, optionally 450 first, 440 second), and in a repetition (line 455) again bit vector defining 425. If during checking 430, interference is detected, YES 431, assigning 460 and end 499 follow. Method 400 continues as long as there are interfaces available to perform bit vector defining step 425, or otherwise terminates (not illustrated for convenience).

In numbering classes step 410, computer 200 determines from the source code which classes use only a single interface (e.g., classes 1 and 4) and which classes use two or more interfaces (multiple interfaces classes, MICs, cf. above introduced equations (1)) and assigns indices j=0 to J-1 to the MICs. The following indices are used in the example:

$$\begin{aligned} &\text{class } 2 \rightarrow \text{MIC } 0 && j = 0 \qquad (3)\\ &\text{class } 3 \rightarrow \text{MIC } 1 && j = 1\\ &\text{class } 5 \rightarrow \text{MIC } 2 && j = J\text{-}1 = 2 \end{aligned}$$

In bit vector defining step 420, 425, computer 200 provides a bit vector B(i) having J bits b(i, j) for interface i according to:

$$B(i) = \{\, b(i,0), \ldots b(i,j), \ldots b(i,J) \,\} \qquad (4)$$

$$\begin{aligned} b(i,j) &= 1 \text{ if MIC } j \textit{ implements} \text{ interface } i\\ &= 0 \text{ otherwise} \end{aligned}$$

Computer 200 can obtain the bit vectors from the source code. The bit vectors (bit triples, J = 3) for the example of FIG. 2 are:

B(1) = { 1, 0, 1 } (for interface 1) (5)

B(2) = { 1, 1, 0 } (for interface 2)

B(3) = { 0, 1, 0 } (for interface 3)

B(4) = { 1, 0, 0 } (for interface 4)

B(5) = { 0, 0, 1 } (for interface 5)

In B(1), the first bit is "1" because MIC 0 (class 2, cf. statement (22) in FIG. 2) uses interface i = 1; the second bit is "0" because MIC 1 (class 3, cf. statement (23)) does not use interface i = 1; and the third bit is "1" because MIC 2 (class 5, cf. statement (25)) uses interface i = 1. In FIG. 8, reference number 420 is used for an initial bit vector defining (e.g., of B(1)), and number 425 is used for defining second and further bit vectors (e.g., B(2), B(3)). It is understood, that once a bit vector has been defined, the bit vector is conveniently stored in memory (e.g., memory 202) so that a recalculation is not required.

In interference checking step 430, computer 200 compares at least two bit vectors B(i) and B(i') and provides check vector *Check* (i, i') by applying logical ***and***-operations to corresponding bits b with equal j-index, that is:

Check (i, i') = B(i) ***and*** B(i') (6)

Check (i, i') = { [b(i, 0) ***and*** b(i', 0)],

...

[b(i, j) ***and*** b(i', j)],

...

[b(i, J) ***and*** b(i', J)] }

In the example, the ***and***-operation provides for B(1) and B(2):

Check (1, 2) = { [1 ***and*** 1] [0 ***and*** 1] [1 ***and*** 0] } (7)

= { 1, 0, 0 }

By checking, computer 200 answers the questions "Interference possible?". For at least a single occurrence of a "1" in Check (i, i'), there would be a possible interference (line YES 431). When all elements of Check (i, i') are "0", there would be no interference (line NO 432), that is, for example,

Check (1, 3) = { [1 ***and*** 0] [0 ***and*** 1] [1 ***and*** 0] } (8)

= { 0 0 0 }

Although explained here for only two bit vectors, it is understood that checking 430 can be applied for any number of bit vectors and that checking 430 (and step 425) can be repeated until no more bit vectors defined in the previous step 425 remain. For simplicity, FIG. 8 does not illustrate repetitions of step 430. As mentioned above, interference checking step 430 and bit vector defining step 425 form interface set identifying step 435 provide the set of non-interfering interfaces. In other words, each time method 400 reaches line 432, computer 200 has identified a new interface set. Persons of skill in the art can arrange this without the need of further explanation herein.

In offset assigning step 440, 460 computer 200 assigned an equal offset to all interfaces of a set or to a single interface, as for example:

$$\textit{Offset} := 0 \text{ for interfaces 1 and 3} \qquad (9)$$

Assigning can be preliminary so that, optionally, consecutive performed assigning steps overcome the old assignment. In FIG. 8, reference number 440 is used for the assigning step which follows the detection of interface sets; and number 460 is used for the assigning step for an offset of the last remaining interface. The repetition of step 440 comprises that the value for *Offset* is incremented for each assignment. Persons of skill in the art can take the necessary measures without the need of further explanation herein.

In merging step 450, computer 200 combines all bits b(i, j) and b(i', j) of first bit vectors B(i) and second bit vector B(i') by logical *or*-operations and provides a new bit vector B°(i) which is identified by the index "i" (same as the first bit vector) and the superscript circle ° symbol. Preferably, the first bit vector B(i) is no longer used so that, preferably, B°(i) overrides B(i). However, this is not essential. If merging is applied to a bit vector B°(i) with one circle °, then the new bit vector is identified by a further circle B°°(i) and so forth. The *or*-operation is applied bitwise, that is:

$$\begin{aligned} B^{\circ}(i) &= B(i) \textit{ or } B(i') \qquad (30)\\ &= \{ \; [\, b(i, 0) \textit{ or } b(i', 0) \,],\\ &\quad\; [\, b(i, j) \textit{ or } b(i', j) \,],\\ &\quad\; \ldots\\ &\quad\; [\, b(i, J) \textit{ or } b(i', J) \,] \; \} \end{aligned}$$

In the example:

$$B^{\circ}(1) = B(1)\ or\ B(3) = \{\ [1\ or\ 0]\quad [0\ or\ 1]\quad [1\ or\ 0]\ \} = \{\ 1\ 1\ 1\ \} \tag{31}$$

Similar as the above checking step, merging step 450 can be applied to any number of bit vectors. In this case, the results of multiple *or*-operations are accumulated. Preferably, merging is applied to the vectors of that interfaces which are identified in step 435.

Referring to the example of FIG. 2, method (400) can be performed as follows: numbering 410; defining B(1) using source code 420; defining B(2) using source code 425; checking 430 B(1), B(2), "Interference?" YES, in repetition defining 425 B(3) using source code checking 430 B(1), B(3), *Check* (1, 3) = { 0 0 0 }, "Interference?" NO; assigning *Offset* = 0 to the interface set of interfaces 1 and 3; merging 450 B(1), B(3) to B°(1) (becoming an "no interference bit vector" for interfaces 1 and 3); continuing at steps 425 and 430 wherein B°(1) is checked with B(4) and so on.

Further, method 400 of the present invention can also be explained as follows. By executing numbering step 410 and defining step 420 for all B(1) to B(5), computer 200 stores bit matrix { B }having bit vector B(i) represented in columns. Matrix { B } has J rows (e.g., 3) and comprises bits b(i, j) as described above. The following representation by columns and row is convenient, but not essential, those of skill in the art can transform the matrix by replacing columns and rows.

$$\{B\} = \begin{array}{ccccc} B(1) & B(2) & B(3) & B(4) & B(5) \\ 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 \end{array} \tag{32}$$

Computer 200 now swaps column B(3) and B(2), thus storing a modifying the matrix, that is

$$\{B\}\# = \begin{array}{ccccc} B(1) & B(3) & B(2) & B(4) & B(5) \\ 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 \end{array} \tag{33}$$

The # symbols stands for the modification. By applying checking step 430, computer 200 now obtains submatrix B(1) B(3) for which *Check* (1, 3) = { 0 0 0 }. By further swapping columns and checking steps, the matrix has been ordered to

$$\{ B \}\#\#= \begin{array}{ccccc} B(1) & B(3) & B(2) & B(5) & B(4) \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \end{array} \qquad (34)$$

{ B } ## comprises submatrices B(1) B(3) and B(2) B(5) standing for interface sets each comprising non-interfering interfaces. In other words, by swapping, computer 200 identifies interface sets (cf. step 435). By assigning step 440, PFT offsets are for each submatrix, that is *Offset* =0 for submatrix B(1) B(3), *Offset* = 1 for submatrix B(2) B(5), and Offset = 2 for submatrix B(4).

The present invention can also be described as an apparatus (e.g., host computer 100) to provide addresses (*Offset, c*) for PFTs in run-time structures (e.g., memory section 120) pointing (e.g., pointer 122) to target computer instructions (e.g., instructions 111), the apparatus receiving code (e.g., source code 240 in Java with an application) indicating which program segments (e.g., classes) implement which interface (i.e., set of functions) and providing addresses each having a first portion (e.g., index "c" for class) corresponding to the program segment and a second portion (e.g., *Offset*) which is equal to all interfaces of the same type throughout the segments, the apparatus characterized in that the apparatus provides addresses with equal second portions also for interfaces of different types which are implemented by the classes in a non-overlapping assignment.

The present invention can also be described as a storage medium (102) for storing Java run-time structures (120) with memory locations (127) storing pointers (PFT, 129), wherein (a) the pointer point to functions (112) belonging to interfaces, (b) some of the interfaces are implemented by classes, and (c) the memory locations are identified by a first index representing the class and a second index representing an offset (*Offset*) within a set (RTS(c)) of memory locations for each class. The storage medium (102) id characterized in that some of the memory locations for different interfaces (e.g., interfaces 1 and 3) share common second indices (e.g., Offset = 0)

when the functions (112) pointed to by the pointers (129) in the memory locations are not commonly implemented by two classes.

While the invention has been described in terms of particular structures, devices and methods, those of skill in the art will understand based on the description herein that it is not limited merely to such examples and that the full scope of the invention is properly determined by the claims that follow.

Glossary of terms

In the following, abbreviations, and writing conventions are listed alphabetically. This glossary is provided only for convenience.

b(i, j)	bit
B(i)	bit vector
B°(i)	new bit vector
{ B }	matrix
Check (i, i')	check vector
f	index for functions
F	number of functions, e.g., F = 2 for all interfaces i
i	index for interfaces
I	number of interfaces, e.g., I = 5
c	index for classes
C	number of classes, e.g., C = 6
$I_{CLASS}(c)$	number of implemented interfaces in a class
j	index for multiple interface classes (MCI)
J	number of MIC
MIC	multiple interface class
Offset	offset to identify PFTs in the run-time structure
PFT	pointer to function table
S, S'	number of slots
X	variable

Claims

1. A method for providing a run-time structure for an application to a computer memory, the method comprising the steps of:

reading a code representation of said application in which interfaces are used by classes, thereby identifying at least a first set of first and second interfaces such as there is no class which uses together said first and second interfaces of said first set; and

writing, separate for each class, pointer to function tables (PFT) to memory locations which are identified by offsets within the classes, thereby assigning a first common offset to corresponding first and second PFTs which belong to said first and second interface, respectively, of said first set.

2. The method of claim 1 characterized by repeating reading and writing for a second set of interfaces with a second common offset so that the smallest offset is assigned to that set which is used by most of the classes.

3. The method of claim 1 wherein identifying comprises determining first elements corresponding to the interfaces regardless to classes, associating some of said first elements by second elements to indicate the use of said first and second interfaces by said first class, and obtaining the first set from these first elements which are not associated by said second elements.

4. The method of claim 3 wherein said first elements are nodes of an interference graph and said second elements are edges between nodes of said graph.

5. The method of claim 1 wherein identifying comprises solving a graph coloring problem.

6. The method of claim 1 wherein said offsets are integer numbers.

7. The method of claim 1 wherein further sets are identified and further offsets are assigned and the smallest offset is assigned to that set which has most interfaces.

8. The method of claim 1 applied to Java applications.

9. The method of claim 1 wherein identifying comprises to define bit vectors for interfaces which indicate interface occurrence in multiple interface classes, and to compare bit vectors at least two interfaces by logical operations.

10. A method for operating a computer system to provide an optimized memory allocation for Java functions, the method comprising the steps of:

reading a Java program with classes and interfaces having functions within the classes to establish an interference graph with nodes for each interface and inter-node edges for the occurrence of first and second interfaces within a single class;

assigning unique offsets to pluralities of interfaces which are not connected by said edges; and

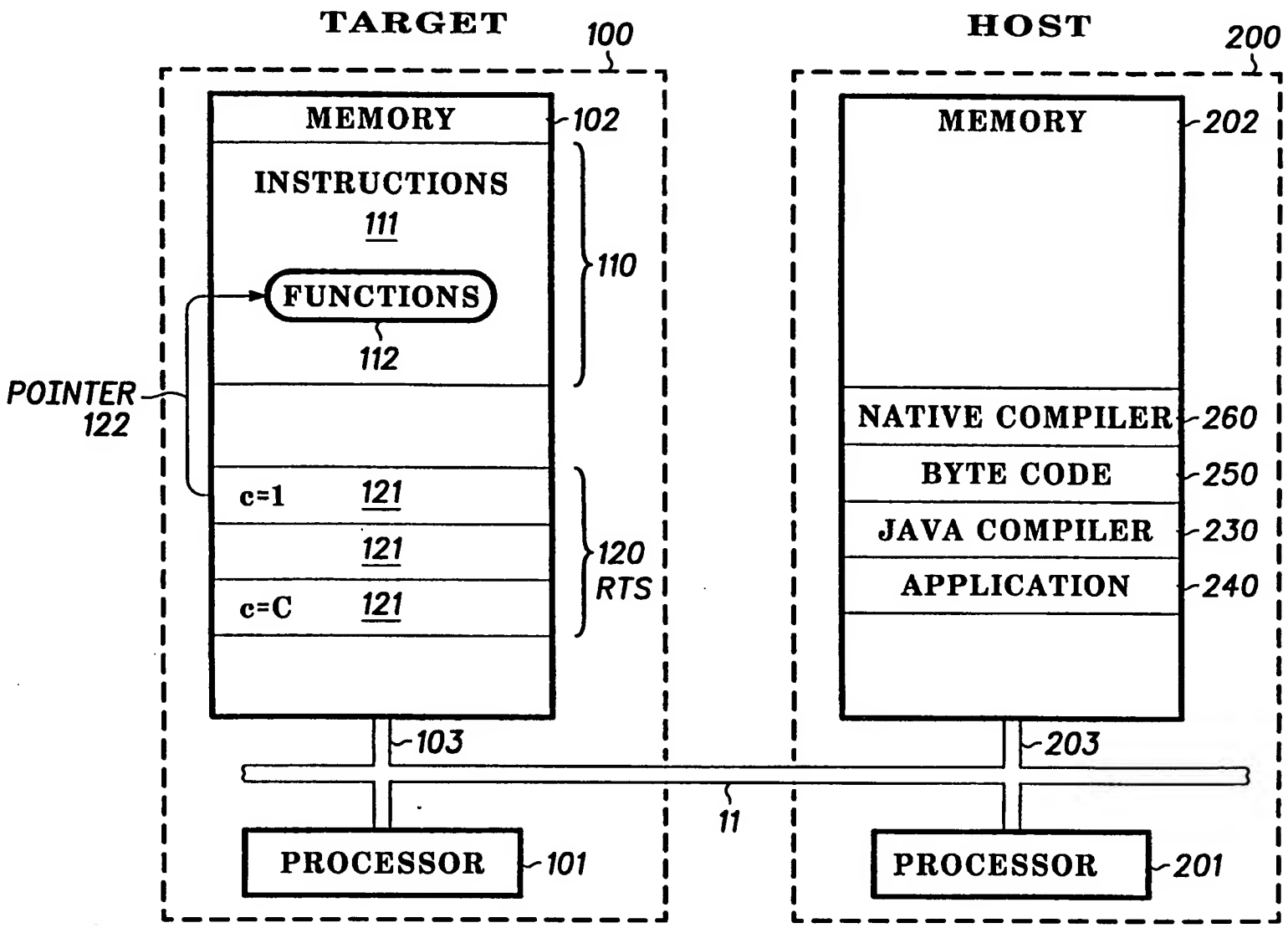
storing pointers to function tables in the memory in sections for each class wherein each pointer in the plurality is identified by the previously assigned offset.

11. The method of claim 10 wherein the assigning step comprises solving a graph-coloring problem.

12. The method of claim 10 wherein said native instructions are identified via function tables.

13. An apparatus to provide addresses for pointers in run-time structures pointing to target computer instructions, said apparatus receiving an assignment code indicating which program segments implement which interface and providing addresses each having a first portion corresponding to the program segment and a second portion which is equal to all interfaces of the same type throughout the segments, said apparatus characterized in that said apparatus provides addresses with equal second portions also for interfaces of different types which are implemented by the classes in a non-overlapping assignment.

14. A storage medium for storing Java run-time structures with memory locations storing pointers, wherein the pointer point to functions belonging to interfaces, some of the interfaces are implemented by classes, and the memory locations are identified by a first index representing the class and a second index representing an offset within a set of memory locations for each class, the storage medium characterized in that some of the memory locations for different interfaces share common second indices when the functions pointed to by the pointers in the memory locations are not commonly implemented by two classes.


TARGET
100
HOST
200
MEMORY
102
INSTRUCTIONS
111
FUNCTIONS
112
110
POINTER
122
c=1
121
121
c=C
121
120
RTS
103
MEMORY
202
NATIVE COMPILER
260
BYTE CODE
250
JAVA COMPILER
230
APPLICATION
240
203
11
PROCESSOR
101
PROCESSOR
201

10 FIG. 1

INTERFACE 1	{ (11)
FUNCTION 11 ( ), FUNCTION 12 ( )	}
INTERFACE 2	{ (12)
FUNCTION 21 ( ), FUNCTION 22 ( )	}
INTERFACE 3	{ (13)
FUNCTION 31 ( ), FUNCTION 32 ( )	}
INTERFACE 4	{ (14)
FUNCTION 41 ( ), FUNCTION 42 ( )	}
INTERFACE 5	{ (15)
FUNCTION 51 ( ), FUNCTION 52 ( )	}
CLASS 1 IMPLEMENTS INTERFACE 1	{ (21)
FUNCTION 11 ( ), FUNCTION 12 ( )	}
CLASS 2 IMPLEMENTS INTERFACE 1,2,4	{ (22)
FUNCTION 11 ( ), FUNCTION 12 ( ),	
FUNCTION 21 ( ), FUNCTION 22 ( ),	
FUNCTION 41 ( ), FUNCTION 42 ( )	}
CLASS 3 IMPLEMENTS INTERFACE 2,3	{ (23)
FUNCTION 21 ( ), FUNCTION 22 ( ),	
FUNCTION 31 ( ), FUNCTION 32 ( )	}
CLASS 4 IMPLEMENTS INTERFACE 3	{ (24)
FUNCTION 31 ( ), FUNCTION 32 ( )	}
CLASS 5 IMPLEMENTS INTERFACE 1,5	{ (25)
FUNCTION 11 ( ), FUNCTION 12 ( ),	
FUNCTION 51 ( ), FUNCTION 52 ( )	}
CLASS 6 IMPLEMENTS INTERFACE 4	{ (26)
FUNCTION 41 ( ), FUNCTION 42 ( )	}

FIG. 2

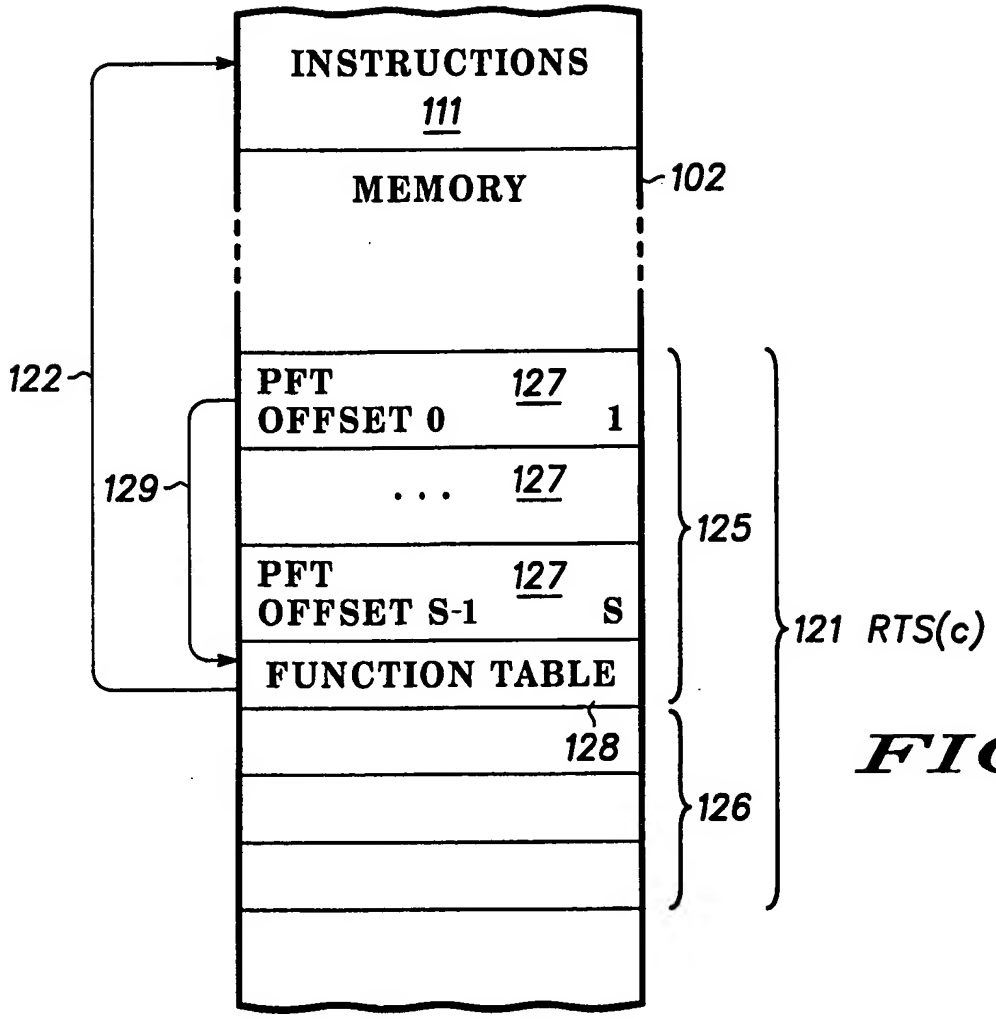
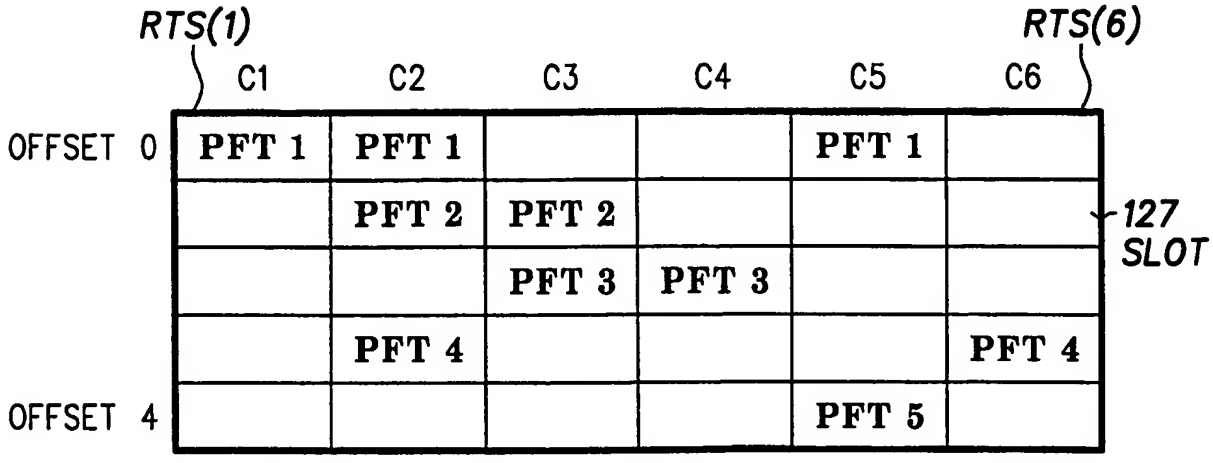


FIG. 3



- PRIOR ART -

FIG. 4

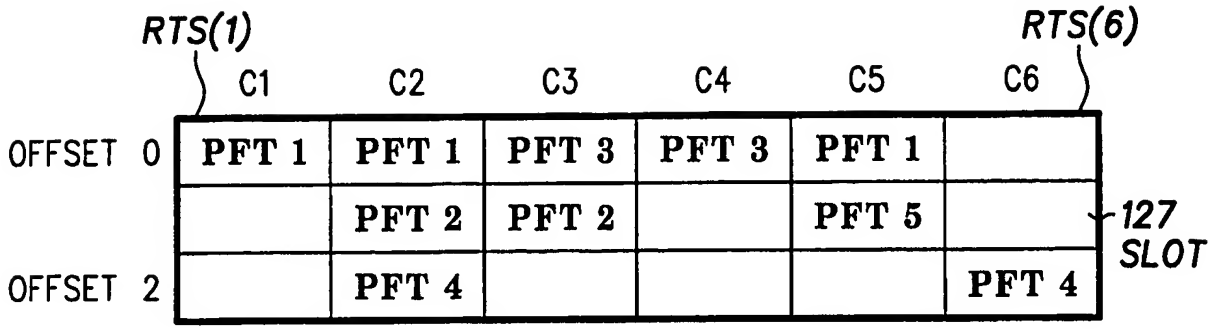


FIG. 5

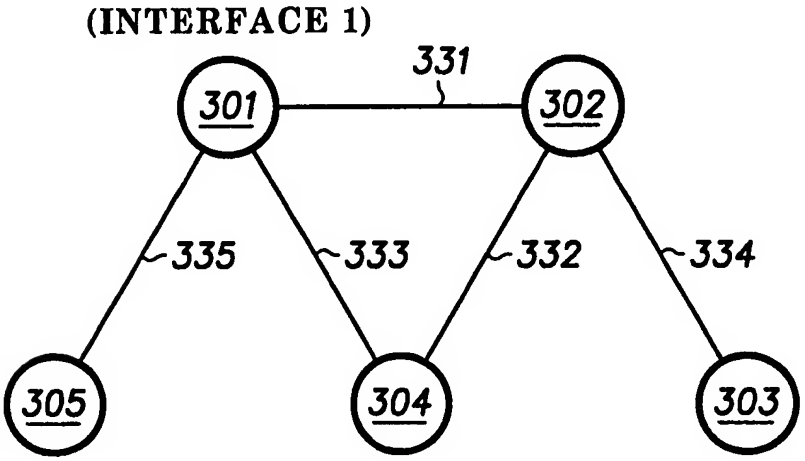


FIG. 6 300

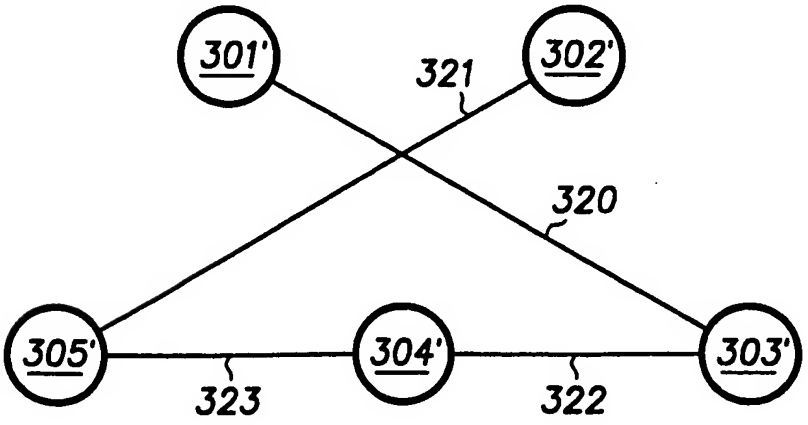
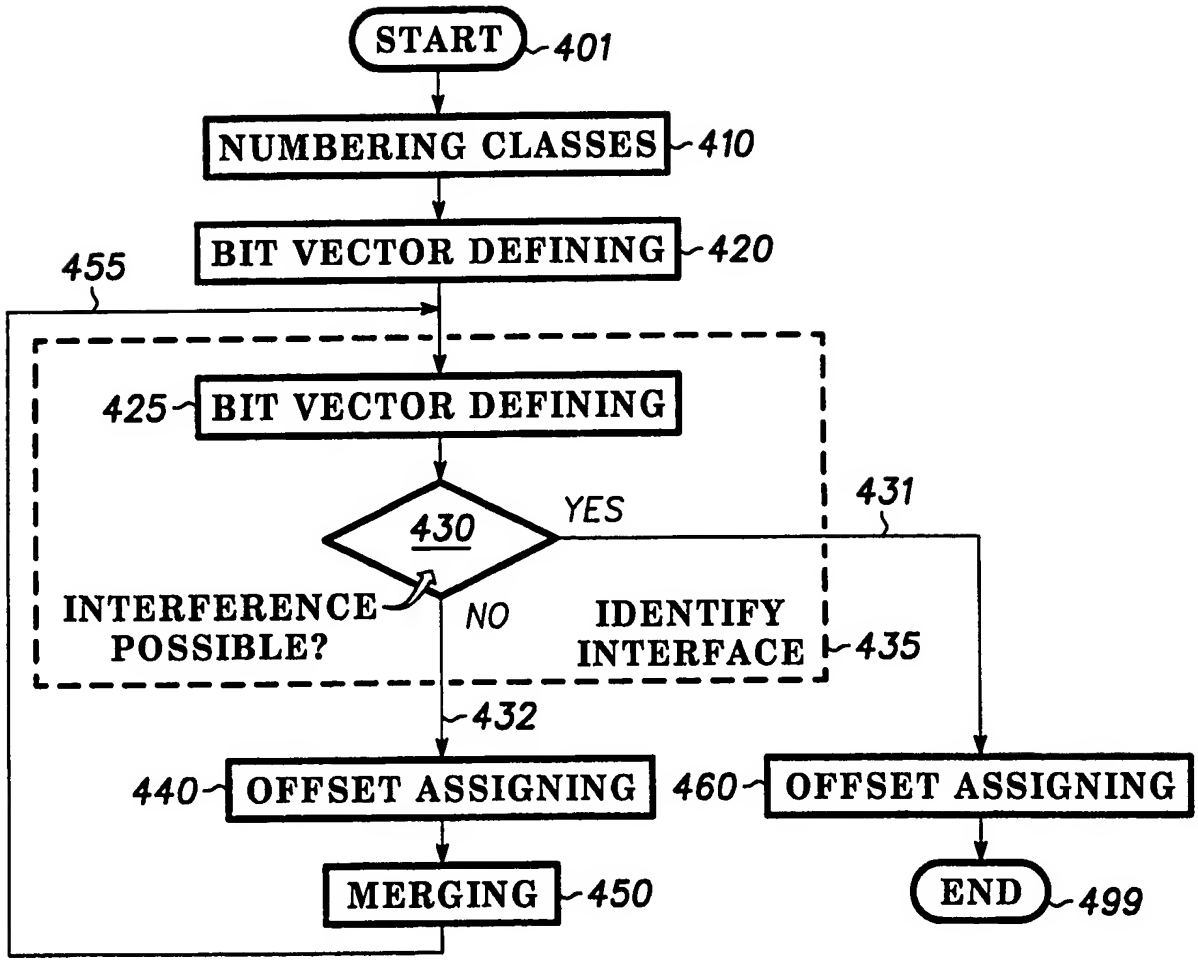


FIG. 7 300'


START
401
NUMBERING CLASSES
410
BIT VECTOR DEFINING
420
455
425
BIT VECTOR DEFINING
430
YES
431
INTERFERENCE POSSIBLE?
NO
IDENTIFY INTERFACE
435
432
440
OFFSET ASSIGNING
460
OFFSET ASSIGNING
MERGING
450
END
499

400

FIG. 8

INTERNATIONAL SEARCH REPORT

Int[illegible]ional Application No
PCT/RU 99/00210

A. CLASSIFICATION OF SUBJECT MATTER
IPC 7 G06F9/44

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)
IPC 7 G06F

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practical, search terms used)

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category *	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
A	"DYNAMIC INTERCEPTION OF IMPORTED PROCEDURE CALLS" IBM TECHNICAL DISCLOSURE BULLETIN,US,IBM CORP. NEW YORK, vol. 39, no. 1, 1 January 1996 (1996-01-01), pages 197-201, XP000556373 ISSN: 0018-8689 the whole document	1-14
A	US 5 854 931 A (JONES DAVID T ET AL) 29 December 1998 (1998-12-29) column 3, line 56 -column 4, line 34	1-14
A	US 5 794 041 A (LAW THEODORE CHEUK-TAK ET AL) 11 August 1998 (1998-08-11) column 2, line 36 -column 3, line 14	1-14

☐ Further documents are listed in the continuation of box C.

☒ Patent family members are listed in annex.

* Special categories of cited documents :

"A" document defining the general state of the art which is not considered to be of particular relevance

"E" earlier document but published on or after the international filing date

"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)

"O" document referring to an oral disclosure, use, exhibition or other means

"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention

"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone

"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art.

"&" document member of the same patent family

Date of the actual completion of the international search	Date of mailing of the international search report
24 May 2000	31/05/2000

Name and mailing address of the ISA	Authorized officer
European Patent Office, P.B. 5818 Patentlaan 2 NL – 2280 HV Rijswijk Tel. (+31–70) 340–2040, Tx. 31 651 epo nl, Fax: (+31–70) 340–3016	Brandt, J

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
US 5854931 A	29-12-1998	US 5754862 A	19-05-1998
		US 5410705 A	25-04-1995
		US 5297284 A	22-03-1994
US 5794041 A	11-08-1998	NONE	